비균일 단일루프에서의 병렬화

Parallelism for Single Loops with Non-uniform Dependences

정삼진
백석대학교

Jeong Sam-Jin
Baeksuk Univ.

요약

본 논문은 비균일 단일루프의 병렬성을 향상시키기 위해서 지금까지 개발된 Threshold 분할 방법, Polychronopoulous 분할 방법과 같은 루프 분할 방법을 소개한다. 그리고, 비균일 단일 루프에서의 병렬화를 최대화 할 수 있는 향상된 루프 분할 방법을 제시한다. 최초로 존재하는 종속성과 본 논문에서 제안한 정의들을 이용하여 비균일 단일 루프를 위한 일반적이면서 최적의 알고리즘들을 제안한다. 제안한 알고리즘들은 일반적인 단일 루프를 병렬화된 루프로 변환하는 방법을 적용한다.

Abstract

This paper reviews some loop partitioning techniques such as loop splitting method by thresholds and Polychronopoulos' loop splitting method for exploiting parallelism from single loop which already developed. We propose improved loop splitting method for maximizing parallelism of single loops with non-constant dependence distances. By using the distance for the source of the first dependence, and by our defined theorems, we present generalized and optimal algorithms for single loops with non-uniform dependences. The algorithms generalize how to transform general single loops into parallel loops.

I. Introduction

Computationally expensive programs spend most of their time in the execution of DO-loops. Therefore, an efficient approach for exploiting potential parallelism is to concentrate on the parallelism available in loops in ordinary programs.

Through some loop transformations using a dependence distance in single loops[1][5], a loop can be splitted into partial loops to be executed in parallel without violating data dependence relations, that is, the size of distance can be used as a reduction factor[1], which is the number of iterations that can be executed in parallel. In the case of non-constant distance such that it varies between different instances of the dependence, it is much more difficult to maximize the degree of parallelism from a loop.

Partitioning of loops requires efficient and exact data dependence analysis. We can consider some tests that examine the dependence of one-dimensional subscripted variables the separability test, the GCD test and the Banerjee test[3].

When we consider the approach for single loops, we can review two partitioning techniques proposed in [1] which are fixed partitioning with minimum distance and variable partitioning with ceil(d(i)). However, these leave some parallelism unexploited, and the second case has some constraints.

The rest of this paper is organized as follows. Chapter two describes our loop model, and introduces the concept of data-dependence computation in actual programs. In chapter three, we review some partitioning techniques of single loops such as loop splitting method by thresholds and Polychronopoulos' loop splitting method. In chapter four, we propose a generalized and optimal method to make the iteration space of a loop into partitions with variable sizes. Finally, we conclude in chapter five with the direction to enhance this work.

II. Program Model and Data Dependence Computation

For data-dependence computation in actual programs, the most common situation occurs when we are

comparing two variables in a single loop and those variables are elements of a one-dimensional array, with subscripts linear in the loop index variable. Then this kind of loop has a general form is shown if Fig. 1. Here, l, u, a_1, b_1, and b_2 are integer constants known at compile time.

```
DO I = l, u
S1 :   A(a1*I + a2) = ···
S2 :          ··· = A(b1*I + b2)
END
```

▶▶ **Fig. 1 A single loop model.**

For dependence between statements S_1 and S_2 to exist, we must have an integer solution (i, j) to equation (1) that is a linear diophantine equation in two variables. The method for solving such equations is well known and is based on the extended Euclid's algorithm[2].

$$a_1 i + a_2 = b_1 j + b_2 \text{ where } l \le i, j \le u \qquad (1)$$

This equation may have infinitely many solutions (i, j) given by a formula of the form:

$$(i, j) = ((b_1/g)t + i_1, (a_1/g)t + j_1 \qquad (2)$$
$$\text{where } (i_1, j_1) = ((b_2-a_2)i_0/g, (b_2-a_2)j_0/g)$$

i_0, j_0 are any two integers such that $a_1 i_0 - b_1 j_0 = g(\gcd(a_1, b_1))$ and t is an arbitrary integer[3][4]. Acceptable solutions are those for which $l \le i, j \le u$, and in this case, the range for t is given by

$$\max(\min(\alpha,\beta), \min(\gamma,\delta)) \le t \le \min(\max(\alpha,\beta), \max(\gamma,\delta))$$
$$\text{where } \alpha = -(l - i_1)/(b_1/g),\ \beta = -(u - i_1)/(b_1/g),$$
$$\gamma = -(l - j_1)/(a_1/g),\ \delta = -(u - i_1)/(a_1/g). \qquad (3)$$

III. Related Works

Now, we review some partitioning techniques of single loops. We can exploit any parallelism available in such a single loop in Fig. 1, by classifying the four possible cases for a_1 and b_1, coefficients of the index variable I, as given by (4).

(a) $a_1 = b_1 = 0$
(b) $a_1 = 0, b_1 \neq 0$ or $a_1 \neq 0, b_1 = 0$
(c) $a_1 = b_1 \neq 0$
(d) $a_1 \neq 0, b_1 \neq 0, a_1 \neq b_1$ (4)

In case 4(a), because there is no cross-iteration dependence, the resulting loop can be directly parallelized. In the following subsections, we briefly review several loop splitting methods for the cases of 4(b) through 4(d).

1. Loop splitting by thresholds

A threshold indicates the number of times the loop may be executed without creating the dependence. In case 4(b), for a dependence to exist, there must be an integer value i of index variable I such that $b_1 * i + b_2 = a_2$ (if $a_1 = 0$) or $a_1 * i + a_2 = b_2$ (if $b_1 = 0$) and $l \le i \le u$. If there is no solution, then there is no cross-iteration dependence and the loop can also be parallelized. And if integer exists, then there exist a flow dependence (or anti-dependence) in the range of I, $[l, u]$ and an anti-dependence (or flow dependence) in $[i, u]$. In this case, be breaking the loop at the iteration $I = i$ (called *turning threshold*), the two partial loops can be transformed into parallel loops.

In case 4(c), let (i, j) be an integer solution to (1), then there exists a dependence in the range of I and the dependence distance (d) is $|j - i| = |(a_2 - b_2)/a_1|$. Here, the loop can be transformed into two perfectly nested loops; a serial outer loop with stride d (called *constant threshold*) and a parallel inner loop[5].

In case 4(d), an existing dependence is non-uniform since there is a non-constant distance, that is, such that it varies between different instances of the dependence. And we can consider exploiting any parallelism for two cases when $a_1 * b_1 < 0$ and $a_1 * b_1 > 0$. Suppose now that $a_1 * b_1 < 0$. If (i, i) is a solution to (1), then there may be all dependence sources in (l, i) and all dependence sinks in $[i, u]$. Therefore, by splitting the loop at the iteration $I = i$ (called *crossing threshold*), the two partial loops can be directly parallelized[5].

2. Polychronopoulos' loop splitting

We can also consider exploiting any parallelism for the case 4(d) when $a_1 * b_1 \geq 0$. We will consider three cases whether it exists only flow dependence, anti-dependence, or both in the range of I. First, let (i, j) be an integer solution to (1). If the distance, $d(i)$ depending on i, as given by (5), has a positive value, then there exists a flow dependence, and if $d_a(j)$ depending on j, as given by (6), has a positive value, then there exists an anti-dependence. Next, if (x, x) is a solution to (1) (x may not be an integer.), then $d(x) = d_a(x) = 0$ and there may exist a flow (or anti-) and an anti-dependence (or flow) before and after I = ceil(x), and if x is an integer, then there exists a loop-independent dependence at $I = x$. Here, suppose that Then for each value of I, the element $A(a_1*I + a_2)$ defined by that iteration cannot be consumed before ceil(d(i)) iterations later, and this indicates that ceil(d(i)) iterations can execute in parallel.

$$d(i) = j-i = D(i)/b_1, \text{ where } D(i) = (a_2-b_1)*i+(a_2-b_1) \quad (5)$$

$$d_a(j) = i-j = D_a(j)/a_1, \text{ where } D_a(j) = (b_1-a_1)*j+(b_2-a_2) \quad (6)$$

```
DO I =1, N
S1 :  A(3I+1) = ···
S2 :     ··· = A(2I-4)
ENDDO
```

▶▶ **Fig. 2 An example of a single loop.**

Consider the loop, as given in Fig. 2, in which there exist flow dependences. $d(i) = D(i)/b_1 = (i+5)/2 > 0$ for each value of I and d(i) have integer values, 3, 4, 5, ··· as the value of I is incremented.

IV. Maximizing Parallelism for Single Loops

From a single loop with non-constant distance such that it satisfies the case (d) in (4) and $a_1*b_1 > 0$, we can get the following theorems. For convenience' sake, suppose that there is a flow dependence in the loop.

Theorem 1: The number of iterations between a dependence source and the next source, sd is given by $|b_1|/g$ iterations where $g = \gcd(a_1, b_1)$.

Proof: Let i, i' be iterations for a source and next source, respectively. Then from (2), $i = (b_1/g)t + i_1$ and $i' = (b_1/g)(t+1)+i_1$ for i_1, g and t, as defined in (2). Therefore, $sd = |i'-i| = |b_1|/g$.

And we can know the facts that if we obtain the iteration for the source of the first dependence, then we can compute the others easily, and $i \equiv j(\text{mod } s_d)$ for i, j are arbitrary iterations for all sources.

Theorem 2: The dependence distance, that is, the number of iterations between the source and the sink of a dependence, is $D(i)/b_1$ where $D(i) = (a_1 - b_1) * i + (a_2 - b_2)$, and the increasing rate of a distance per one iteration, d′ is given by $(a_1 - b_1)/ b_1$. And the difference between the distance of a dependence and that of the next dependence, d_{inc} is $|a_1 - b_1|/g$.

Proof: According to (5), we can know the distance and d′. And $d_{inc} = d' * s_d = (a_1 - b_1)/ b_1 * |b_1|/g = |a_1 - b_1|/g$.

Hence, if we obtain the distance for the source of the first dependence, then we can compute the others easily. Also for the case of an anti-dependence, similarly, Theorem 1 and 2 can be represented. Namely, s_d is given by $|a_1|/g$ iterations where $g = \gcd(a_1, b_1)$, the distance is given by (6), and d′ is $(b_1-a_1)/a_1$. And $d_{inc} = d' * s_d = (b_1-a_1)/a_1 * |a_1|/g = |b_1 - a_1|/g$.

By using the iteration and distance for the source of the first dependence, and concepts defined by Theorem 1 and 2, we obtain the generalized and optimal algorithm to maximize parallelism from single loops with non-uniform dependences.

Procedure MaxSplit shows the transformation of single loops satisfying the case (d) in (4) and $a_1*b_1 > 0$ into partial parallel loops.

In step 2 in Procedure MaxSplit, the iteration for the sink of the first dependence in each block, Sri + di, is selected as the first iteration in the next block, St_{i+1}, to maximize parallelism from a loop. And the iteration and distance for the source of the first dependence in each of blocks are obtained in step 3 and 4, respectively. The transformed loop will speed up by a factor of $St_{i+1} - St_i$ (reduction factor, which is the number of iterations that

can be executed in parallel), the stride of the outer loop.

Procedure MaxSplit (l, u, s_d , d_{inc}, α, β)

/*Transformation of single loops with non-uniform dependences into partial parallel loops */

BEGIN

/* (1) Computing the first iteration in each of blocks partitioned by loop splitting.

St_i : the first iteration in the ith block

Sr_i : the iteration for the source of the first dependence in the ith block

d_i : the distance for Sr_i

dn_i : the sequential number of the source of the first dependence in the ith block with respect to all sources in the original loop

l, u: the lower and upper bounds of loop, respectively

α, β: the iteration and distance for the source of the first dependence in the loop computed by the separability test, respectively

s_d, d_{inc}: the same as defined in theorem 1 and 2, respectively */

Step 1: $I = 1$; $St_1 = l$; $Sr_1 = \alpha$; $d_1 = \beta$;

Step 2: $St_{i+1} = Sr_i + d_i$;

If $St_{i+1} \geq u$ then {$St_{i+1} = u + 1$ go to Step 5}

Step 3: $Sr_{i+1} = St_{i+1} + q$, where $0 \leq q < s_1$ and $q = (Sr_1 - St_{i+1}) \bmod s_d$;

Step 4: $dn_{i+1} = (St_{i+1} - St_i)/s_d + 1$;

$d_{i+1} = d_1 + (dn_{i+1} - 1) * d_{inc}$

$i = i + 1$ go to step 2;

/*(2) Transforming the original loop into the following parallel loop. */

Step 5: $i = 1$; $I' = l$;

While $I' \leq u$ Do

$inc = St_{i+1} - St_i$;

DOALL $I = I'$, $I' + inc - 1$

$A(a_1*I + a_2) = \cdots$

$\cdots = A(b_1*I + b_2)$

ENDDO

$I' = I' + inc$; $i = i + 1$;

Endwhile

END MaxSplit

Applying Procedure MaxSplit to the loop in Fig. 2, the unrolled version of this result is shown in Fig. 3(c). As shown in Fig. 3, we can know that Procedure MaxSplit is an algorithm to maximize parallelism from single loops with non-constant distances.

I	A(3*I+1)	A(2*I-4)
1	A(04)	A(-2)
2	A(07)	A(00)
3	A(10)	A(02)
4	A(13)	A(04)
5	A(16)	A(06)
6	A(19)	A(08)
7	A(22)	A(10)
8	A(25)	A(12)
9	A(28)	A(14)
10	A(31)	A(16)
11	A(34)	A(18)
12	A(37)	A(20)
13	A(40)	A(22)
14	A(43)	A(24)
15	A(46)	A(26)
16	A(49)	A(28)

(a) Using minimum distance.

I	A(3*I+1)	A(2*I-4)
1	A(04)	A(-2)
2	A(07)	A(00)
3	A(10)	A(02)
4	A(13)	A(04)
5	A(16)	A(06)
6	A(19)	A(08)
7	A(22)	A(10)
8	A(25)	A(12)
9	A(28)	A(14)
10	A(31)	A(16)
11	A(34)	A(18)
12	A(37)	A(20)
13	A(40)	A(22)
14	A(43)	A(24)
15	A(46)	A(26)
16	A(49)	A(28)

(b) Using ceil(d(*i*)).

I	A(3*I+1)	A(2*I-4)
1	A(04)	A(-2)
2	A(07)	A(00)
3	A(10)	A(02)
4	A(13)	A(04)
5	A(16)	A(06)
6	A(19)	A(08)
7	A(22)	A(10)
8	A(25)	A(12)
9	A(28)	A(14)
10	A(31)	A(16)
11	A(34)	A(18)
12	A(37)	A(20)
13	A(40)	A(22)
14	A(43)	A(24)
15	A(46)	A(26)
16	A(49)	A(28)

(c) Our proposed method.

▶▶ **Fig. 3. The unrolled versions of transformed**

V. Conclusions

In this paper, we have studied the parallelization of single loop with non-uniform dependences for maximizing parallelism. For single loops, we can review

two partitioning techniques which are fixed partitioning with minimum distance and variable partitioning with ceil(d(i)). However, these leave some parallelism unexploited, and the second case has some constraints. Therefore, we propose a generalized and optimal method to make the iteration space of a loop into partitions with variable sizes. Our algorithm generalizes how to transform general single loops with one dependence into parallel loops.

In comparison with some previous splitting methods, our proposed methods give much better speedup and extract more parallelism than other methods.

Our future research work is to consider the extension of our method to n-dimensional space.

References

[1] C. D. Ploychronopoulos, " Compiler optimizations for enhancing parallelism and their impact on architecture design," in IEEE Trans. computers, Vol.37, No.8, pp.991-1004, Aug. 1988.

[2] D. E. Knuth, The Art of Computer Programming, vol. 2: Seminumerical Algorithms, Reading, MA: Addison-Wesley, 1981.

[3] H. Zima and B. Chapman, Supercompilers for Parallel and Vector Computers, Addison-Wesley, 1991.

[4] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, " Automatic program parallelization, in Proceedings of the IEEE, Vol.81, No.2, pp.211-243, Feb 1993.

[5] J. R. Allen and K. Kennedy. "Automatic translation of Fortran programs to vector form," in ACM Trans. Programming Languages and Systems, Vol.9, No.4, pp.491-542, Oct. 1987.